IRis

Northeastern University

High Tech Tools & Toys Labwork

High Tech Tools & Toys Lab

January 01, 2004

Lab 6: Back to the Speed of Sound

Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems (Gordon-CenSSIS)

Recommended Citation

Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems (Gordon-CenSSIS), "Lab 6: Back to the Speed of Sound" (2004). *High Tech Tools & Toys Labwork*. Paper 6.

This work is available open access, hosted by Northeastern University.

GE U111 HTT&TL, Lab 6

Back to the Speed of Sound in Air

Introduction to C++ Programming

Contents

1 Preview: Programming & Experiments Goals **2**

2 Part I **2**

2.1 Part I Homework Assignment 2

2.2 Using Visual C++ 4

2.3 Math Primer: A Straight Line Approximation of Data Pairs. 5

2.4 Speed of Sound Data Analysis Using C 6

3 Part II **6**

3.1 Part II Homework Assignment 6

3.2 Speed of Sound Data Analysis Using C, Revisited 7

1 Preview: Programming & Experiments Goals

This session comprises two parts.

Part I is used to introduce basic Visual C++. As a starting point, the experimental context repeats the simple framework of Lab 1. In the programming arena, however, the transition from the interpreted MATLAB to the compiled Visual C++ requires a much more elaborate preparation. Programming concepts and aspects introduced in Part I, include

- Introduction to the Visual C++ environment
- Basic program structure
- Variable (and array) declarations
- Simple input and output commands
- Arithmetic operations

Part II is used to introduce and exercise more advanced Visual C++ tools, to refine the speed of sound computation program, in Part I. Part II uses the experimental results from Lab 1 and Part I of this Lab. Programming concepts and aspects introduced in Part II, include

- Logic functions and branching
- *while* loops in C++
- Input and output files

(Engineering / Science) concepts, components & equipment including

- Same as in Lab 1

Lab Outline. In Part I we shall use experimental data to estimate the speed of sound in air and the correction terms, as in Lab 1, using very simple programming tools (screen input and output and basic arithmetic operations). In Part II we shall revisit this same task, having learned how to read and write data files and include loops in programs.

2 Part I

♣ All reading and Pre-Lab assignment from this session on relate to the book Applications Programming in C++.

♠ As Usual, all pre-lab reports must include copies of -

- Programs you prepare
- Printout of screen output (see instructions in §2.2).
- Output files, where applicable (not this Lab).
- Plots, where applicable.

2.1 Part I Homework Assignment

Assignments for Part I are due in the first session of Part I -

1. Read Chapter 1, in Applications Programming in C++.
2. Read this handout carefully. (§2.2 is essential for your programming tasks.)
3. Do the following in a Pre-Lab assignments:

- Exercise 1 and 3, pages 34-35; Exercises 1 and 2, page 46; Exercise 1,2,3 and 4, page 49
- Do the Pre-Lab Task 1, in this handout.

The assignments above should be included in the Pre-Lab report for Part I.

4. The Pre-Lab assignments for Part II are specified in §3, and will be submitted separately, in the first session of Part II.

5. Only one Post-Lab Report is due, combining Parts I and II. It is due a week after the completion of Part II.

2.2 Using Visual C++

Writing and running a C++ program requires much more than you got used to in MATLAB. For orderly run, each program needs associated with a folder containing, the program itself, input and output data files, and several automatically generated other files, contained in a project. Following is an outline of the steps you have to follow in order to create a new program:

1. Open the Visual C++ Window.
2. Under the `File` menu, select `new`, then`projects`; a dialog box will open -
   - Select `win32 console application`
   - In the `Location` window, select the location of your new project. (In the HTT&TL use the a: directory.)
   - In the `Project name` type a name of your choice.
3. Under the `File` menu, select `new`, then `Files`;
   - `c++ source file`
   - Mark the `Add to project` box, and leave the created project name name showing in the window
   - Leave the default location. (It will be in the sub-directory `Debug` of the directory created for your new project.)
   - In the `Project name` type a name of your choice. Typically it will be the same as the project name. You do not need to include a suffix.
4. Type your program into the file (the top-right window.)
5. `Compile` your program, using one of three options-
   - Above the program / files window are six symbol icons (the right-most being a hand). Click on the left-most icon to compile.
   - Select `compile <file name>` in the `Build` menu.
   - Hit the control and F7 keys together.

   The response and error messages are provided in the bottom window.
6. Once the program is debugged and compiles successfully (no errors or warnings), `Build / Link` your program, using one of three options-
   - Click on the second from left icon above the files window.
   - Select `Build <file name>` or (better yet) `Rebuild all` in the `Build` menu.
   - Hit the F7 key.

   Again, the response and error messages are provided in the bottom window.
7. Once the program is debugged, compiled and linked successfully (no errors or warnings), `Execute` your program, using one of three options-
   - Click on the fourth from left icon above the files window (the `!` icon).
   - Select `Execute <file name>` in the `Build` menu.
   - Hit the control and F5 keys together.
8. Screen input and output in Visual C++ are performed in a DOS style black background window. This window does not have a menu bar. To copy the contents of a portion of that window bring the cursor to the desired beginning point, press the left mouse bottom, drag and highlight the area then release the left right mouse bottom and click the right mouse bottom. Now open a text editor (e.g., notebook) and select the `paste` menu or press `control v`. **Notice:** The order of the mouse bottoms use is essential: a click in the wrong order may close the display window, or display a very long history of past runs.

As you read Chapter 1 you will notice the basic structure of a C / C++ program:

- Calling header files, using the `#include <...>` format.
- Other preliminary global declarations (e.g., `using namespace std`).
- Program and output declaration (e.g., `void main()`).
- The program itself, between curled brackets (i.e., { and }), where each command is followed by a semicolon.

2.3 Math Primer: A Straight Line Approximation of Data Pairs.

Since functions such as MATLAB's `polyfit` and `polyval` are not available in C, we need to create them explicitly. This section provides the details.

Problem Statement.

Given: Pairs of points $(x_i,\, y_i), \qquad i = 1, 2, \ldots, n.$

Postulation: The given pairs are roughly matched by a linear relation

$$\mathtt{y} \;=\; \mathtt{a}\,\mathtt{x} \;+\; \mathtt{b} \tag{1}$$

Task: Find a and b so that the *root mean squares (RMS)* error

$$\epsilon = \sqrt{\sum_{\mathtt{i}=1}^{\mathtt{n}} \left(\mathtt{y_i} - (\,\mathtt{a}\,\mathtt{x_i} + \mathtt{b})\right)^2} \tag{2}$$

will be minimal. A good measure of the quality of the estimation is the normalized error:

$$\epsilon_0 = \frac{\epsilon}{\sqrt{\sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{y_i^2}}} = \sqrt{\frac{\sum_{\mathtt{i}=1}^{\mathtt{n}} \left(\mathtt{y_i} - (\,\mathtt{a}\,\mathtt{x_i} + \mathtt{b})\right)^2}{\sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{y_i^2}}} \tag{3}$$

Problem Solution. This and similar RMS optimization problems are commonplace in such fields as signal processing, communications, medical imaging, radar, control, and stochastic estimation. The theory behind the solution goes beyond the scope of this class, and we shall be content here with the explicit formulae of the solution Procedure.

1. Create two 2×1 vectors M and N, as follows -

$$\mathtt{M(1)} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{x_i}, \quad \mathtt{M(2)} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{x_i^2}, \quad \mathtt{N(1)} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{x(i)y(i)}, \quad \mathtt{N(2)} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{y(i)} \tag{4}$$

2. The optimal values of a and b are

$$\mathtt{a} = \frac{\mathtt{n\,N(1)} - \mathtt{M(1)\,N(2)}}{\mathtt{n\,M(2)} - \mathtt{M(1)^2}}, \qquad \mathtt{b} = \frac{\mathtt{M(2)\,N(2)} - \mathtt{M(1)\,N(1)}}{\mathtt{n\,M(2)} - \mathtt{M(1)^2}} \tag{5}$$

Clearly, in order to code this algorithm efficiently we need the use of both loops - to implement the sum - and logical branching (`if`...) - to avoid division by zero in case $\mathtt{n\,M(2)} = \mathtt{M(1)^2}$. Also, if we want to be able to retrieve data from a file and save the results in a file (e.g., in order to plot the data vs. the approximation), a mechanism to read and write data files is needed. These tools will be introduced in Part II. At this point we shall be content with using only basic arithmetic operations, will trust / hope that zero division will not occur (which is supported by theory, if your measurement distances are amply spaced), and use "manual" data entry and screen output, wit the `cin` and `cout` commands.

2.4 Speed of Sound Data Analysis Using C

Pre-Lab Task 1. Write a C++ program named `line approx.cpp` that executes the following operations:

1. The program calls the header files `iostream` and `cmath`.
2. Here and throughout, use the command `using namespace std`.
3. The program itself has no output; i.e., the program starts with `void main()`.
4. Define `double` arrays named `distance` and `delay`, each of size `5`.
5. Use the `cout` command to display the statement `Please enter values of 5 measured distances between the two transducers` and move the curser to a new line.
6. Use the `cin` command to enter 5 values for `distance[k], k=1,...,5`.
7. Use the `cout` command to display the statement `Please enter corresponding values of 5 measured sound wave travel times between the two transducers` and move the curser to a new line.
8. Use the `cin` command to enter 5 values for `delay[k], k=1,...,5`.
9. Use basic arithmetic operations (i.e., +, -, * and /) to compute the optimal linear approximation of the plot of `distance` as a function of `delay`. That is, code the algorithm of §2.3 with `n=5`, `x=delay` and `y=distance`.
10. Assign the value of the optimal "`a`" to a float variable named `sos` and the value of the optimal "`b`" to a float variable named `displacement`.
11. Compute the normalized error from Eq. (3) to a float variable named `nerror`.
12. Use the `cout` command to display the statement `The estimated speed of sound is <value of sos>`, move to a new line and display the statement `the value of the linear displacement is <value of displacement>`, move to a new line and display the statement `the relative error measure in this estimation is <value of nerror>.` and finally, again, move the cursor to a new line.

Include the program in your Pre-Lab report. Try to debug the program and run it with data from Lab 1. If successful, for extra credit include in your report a printout of the display window from a successful run. Bring a copy of the program on a diskette to the lab.

Lab Task Experiment 1. Use the instructions from Lab 1 to obtain an additional 5 data pairs at nearly equally spaced distances that were not used in your Lab 1 experiment. If needed, use instructor and TA help to debug your program `line approx.cpp`. Run the debugged program using the new experimental data. Include in your report a printout of the display window from a successful run.

3 Part II

3.1 Part II Homework Assignment

1. Review Chapter 1, Sec. 1.9 and read Chapter 2, Sections 2.1, 2.3, 2.5 and 2.8 in Applications Programming in C++.
2. Read this handout carefully.
3. Do the following in a Pre-Lab assignments:
   - Exercise 1, 2, 4, 7 pages 78, 79; Exercise 1, 4, page 86; Exercise 2, pages 95.
   - Do Pre-Lab Task 2 in this handout.
4. Following completion of the Lab, write a Post-Lab report, summarizing the experiments and your findings of both Parts I and II.

3.2 Speed of Sound Data Analysis Using C, Revisited

Pre-Lab Task 2. The program `line approx.cpp` had several disadvantages:

- The program requires manual input and output, instead of a convenient use of data files. In particular, the program output was not readily portable for use with other software, such as the use of MATLAB for plotting.
- The long summation ($\sum_{i=1}^{5}$) required a term by term arithmetic.
- There was no prevention of division by zero.
- The number of data points (5) was pre-determined.

In this task you have to write an improved C++ program named `better line approx.cpp`, building on the original `line approx.cpp`, and improving on it, as follows.

1. The program will read data pairs, one pair at a time, from a data file.
2. The number of data pairs is allowed to vary between 5 and 100. If less than 5 data pairs will be provided, the program will terminate and issue an error statement. If more than 100 data pairs were provided, only the first 100 will be used.
3. The program should stop and issue an error message if either of the denominators in Eqs. (5) and (3) is too small. (Appropriate thresholds can be set at magnitudes lower by a factor of 100 smaller than the values in the computations you made in Part I.)
4. If the threshold and data size conditions are satisfied, the program should continue. The program will then create a MATLAB script file as an output. The M-file will be such that when run in a MATLAB window, it will create the plot created in Lab 1, comparing the linear regression with the data points.
5. In addition, the program should provide for a screen display of the estimated speed of sound and the normalized estimation error.

The following guidelines will help you to create the program:

1. Create a text input file named `mydata.txt`. The file will include a list of experimental data pairs, typed in succession, in the form: <delay> <distance> <newline>. For example:

```
8.2669352e-005 2.0000000e-002
1.5459763e-004 4.0000000e-002
1.7829775e-004 6.0000000e-002
2.6604091e-004 8.0000000e-002
2.6766557e-004 1.0000000e-001
3.9174423e-004 1.2000000e-001
3.9327882e-004 1.4000000e-001
4.5055605e-004 1.6000000e-001
5.5980265e-004 1.8000000e-001
6.0742916e-004 2.0000000e-001
```

2. The program calls the header files `fstream`, `iostream`, and `cmath`. The addition of `fstream` enables the use of input and output files, using the `ifstream` and `ofstream` commands.
3. The program defines the `ifstream` variable `inp` and opens the input file `mydata.txt`. This is done with the commands

```
ifstream inp;
inp.open("mydata.txt");
```

4. The program defines the `ofstream` variable `output` and opens the output file `soscomp.m`. (That is, the output file will be a MATLAB script file that will be run to create desired plots.) This is done with the commands

```
ofstream output;
output.open("soscop.m");
```

5. The program itself has no output; i.e., the program starts with `void main()`.

6. Declare `double` arrays of length 100 `time[100]` and `distance[100]`. These variables will be used to read the respective values of data pairs from the data file.

7. Declare an integer variable `count`, initiated with the value 0. It will be used to count how many input data pairs were read.

8. Declare `double` variables named `M1, M2, N1, N2` and `N3`, each initiated with the value 0. The variables `M1, M2, N1` and `N2` will be used to iteratively compute the respective sums, as defined in Eq. (4). The variable `N3` will be used to iteratively compute the value of the sum $\mathtt{N3} = \sum_{\mathtt{k}=0}^{\mathtt{n}} \mathtt{y}_{\mathtt{k}}^2$, as in the numerator of Eq. (3), where $\mathtt{y}_{\mathtt{k}} = \mathtt{dist}[\mathtt{k}]$.

9. Declare `double` variables named `sos`, `displacement` and `nerror`.

10. When finally created, the output file `soscop.m` should define a two-column MATLAB array named `mydata`, whose first column is the vector of time delays, and the second column is the vector of corresponding distances. These columns will be used in the required plot. Using the numerical example above, the file should eventually include the text

$$
\begin{array}{ll}
\mathtt{mydata} = [& \\
8.2669352e-005 & 2.0000000e-002 \\
1.5459763e-004 & 4.0000000e-002 \\
1.7829775e-004 & 6.0000000e-002 \\
2.6604091e-004 & 8.0000000e-002 \\
2.6766557e-004 & 1.0000000e-001 \\
3.9174423e-004 & 1.2000000e-001 \\
3.9327882e-004 & 1.4000000e-001 \\
4.5055605e-004 & 1.6000000e-001 \\
5.5980265e-004 & 1.8000000e-001 \\
6.0742916e-004 & 2.0000000e-001 \\
]; &
\end{array}
\tag{6}
$$

(Recall that moving to a new line, in a MATLAB definition of an array, is equivalent to entering a semicolon, to instruct starting a new line.) The idea is to use the C++ program to add data values of each pair, as it is read, into the text of the M-file. This will be done below. However, the program must first print into the file the text of the first line and the instruction to move to a new line -

```
output<<"mydata=[\n";
```

11. Create a `while` loop that performs the following steps -

    (a) The `while` condition should combine two basic requirements:

        i. The number of data pairs already read is less than 100.

        ii. There is an additional, unread data pair in the data file. In that case, the first available data entry should be read into `time[count]` and the second available data entry be read into `dist[count]`.

This is done by the command -

```
while ((inp>>time[count])&&(inp>>dist[count])&&(count<100))
```

(b) Write into the output M-file the values of `time[count])` and `dist[count]`, separated by a space, and move to a new line-

```
output<<time[k]<<" "<<dist[k]<<"\n";
```

(c) `time[count]` will be added to the value of `M1`.

(d) `time[count]*time[count]` will be added to the value of `M2`.

(e) `time[count]*dist[count]` will be added to the value of `N1`.

(f) `dist[count]` will be added to the value of `N2`.

(g) `dist[count]*dist[count]` will be added to the value of `N3`.

(h) Finally, having read yet another data pair, the value of `count` has to be increased by one.

By the time this loop terminates, all data pairs (up to 100) were read, the value of `count` is the number of data pairs used, and the values of `M1, M2, N1` and `N2` are as defined in Eq. (4), with `n=count, x=time` and `y=dist`. Thus, Eq. (5) can be used to compute the speed of sound and the displacement correction. Also, the value of `N3` is the numerator in the computation of the relative error in Eq. (3). Finally, all the entries in the definition of the `n`$\times$`2` array `mydata` in the output M-file `soscomp.m` have been written. All that is missing in the definition of `mydata` is the closing line in (6).

12. Following the end of the `while` loop, there should thus be the command

```
output<<"];\n";
```

13. A logical branching should now take the program in one of two directions:

(a) If either $N3$ (the denominator in (3)) or the absolute value of the denominator in (5) is smaller than the respective tolerances that you selected, or if the number of available data pairs is smaller than 5, then

i. A screen error message is issued (using `cout`).

ii. The values of both `sos` and `displacement` are set to the non-physical zero.

iii. The value of the relative error, `nerror`, is set to one (= 100% error).

(b) Else,

i. A screen message of jubilation is issued (using `cout`).

ii. The values of `sos` and `displacement` are computed according to Eq. (5).

iii. The value of `nerror` is computed by Eq. (3). This requires running a loop of `count` steps, to compute the numerator, dividing by `N3` and taking the square root, with the function `sqrt`.

14. Once the values of `sos`, `displacement` and `nerror` have been determined (either way), the program has to include in the output M-file, the following text

```
% The number of data pairs used is <value of count>.

% The estimated speed of sound is <value of sos>.

% The estimated displacement is <value of displacement>.

% The normalized estimation error is <value of nerror>.
```

15. If the value of `nerror` is one or more, it should add to the output file the following text

```
disp('The Experimental Data Are Faulty')
```

16. If the value of `nerror` is less than one, it should add to the output file the following text

```
est=[<value of sos>, <value of displacement>];
count=<value of count>;
nerror=<value of nerror>;
plot(mydata(:,1),mydata(:,2),'*',mydata(:,1),polyval(est,mydata(:,1))
xlabel('travel time')
ylable('distance')
title('A Linear Regression of Distance vs.  Travel Time Data')
```

17. When all tasks are complete, both the input and the output files are closed.

Include the program in your Pre-Lab report.

Lab Task Experiment 2. Debug the program `better line approx.cpp` and run it with 15 data points from Lab 1 and from Part I of this Lab 6. Include in your report a printout of the display window from a successful run. Run the output file in MATLAB. Include in your report a printout of the output MATLAB file `soscomp.m`, the MATLAB command window portion from a successful run and the output plot. Save a copy of the program on a diskette.